Automatic Validation of Concurrent Programming

* * *

Final Report

Christof Bruetsch
CommunicationSystems Student
6th Semester

June 21, 2003

Contents

List of Tables

List of Figures

Part I
Introduction

1 Introduction

The handling with concurrent programs is often a very tricky matter and only hardly solvable. Especially semaphores and locks (verrous in French) offer a wide platform for complicated concurrency situations. Briefly, how can we easily master the complexity of concurrency in programming? The project "Automatic validation of concurrent programs" - in the following called "AVCP" - makes us come a few steps closer.

1.1 Description of the AVCP Project

One possibility that leads to a better understanding of concurrency problems is the generation of state diagrams. Due to the peculiar characteristic of the human brain, pictures, images and charts do often say more than lines in a hyper-complex coded program. So, such diagrams help us to get the point much faster of how a program woks. The project's outline was formulated as follows:

"Develop a tool that parses a given program code and generates all possible states, identifying possible deadlocks and violations of the specifications."

1.2 AVCP - The Plan

Conceptually, there are two main project phases. The first phase consists of designing a new programming language which implements concurrency expressions for semaphores and locks. The main condition is to remain simple and handy, i.e. we would like to have a strongly limited set of grammar rules for this language. Then, after having described the language by a set of grammar rules, it is meant to use a parser generator in order to be able to parse a given code written in this specific matter. I have chosen JavaCC, the JavaCompilerCompiler to generate the parser and JTB, the JavaTreeBuilder for the tree generation - the descriptions of these tools will follow in 2.3.1 and in 2.3.2 later on. The second phase represents the creation of a state diagram. At the end we would like to have a graphical representation of every possible state that can be generated by the given code.

The main tool is written in the Java 2 Programming Language.

Part II
Project Development - The Phases

2 Phase I: The Chrishy Language

Chrishy is the approach to the demanded programming language mentioned and was created with a limited set of grammar rules. In the following, the listing and the explanation of the ChrishyGrammar, written in EBNF: (personal commentary is represented like /* *personal Comment* */)

2.1 Lexical Grammar

/* *the input is an unlimited set of inputelements* */

```
input = { inputelement }
inputelement = whitespace
             | comment
             | token
```

/* *the tokens of Chrishy* */

```
token = ident
      | number
      | string
      | "("
      | ")"
      | "{"
      | "}"
      | ":"
      | ";"
      | ","
```

/* *the (logical) operators* */

```
      | "+"
      | "-"
      | "*"
      | "/"
      | "%"
```

```
            | "="
            | "=="
            | "!="
            | ">"
            | "<"
            | ">="
            | "<="
            | "!"
            | "&"
            | "|"
```

/ important keywords of Chrishy */*

```
            | "procedure"
            | "var"
            | "init"
            | "end"
            | "integer"
            | "semaphore"
            | "verrou"
            | "if"
            | "then"
            | "else"
            | "while"
```

/ important keywords for locks and semaphores */*

```
            | "verrouiller"
            | "deverouiller"
            | "P("
            | "V("
```

/ the commentary */*

```
    comment = "/" "/" { cchar }
```

/ specifications of identifiers, numbers and strings */*

```
    ident = letter { letter | digit | "_" }
    number = digit { digit }
    string = "\"" { schar } "\""
```

/* *the terminal symbols for identifiers, numbers and strings* */

```
whitespace = " " | "\t" | "\f" | "\n"
letter = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i"
       | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r"
       | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"
       | "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I"
       | "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R"
       | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z"
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

/* *string and commentary specifications are just pointed out literally. The grammatical formulation is straightforward* */

```
cchar = every character except "\n"
schar = every character except "\n" and "\""
```

2.2 Syntactical Grammar

/* *a Chrishy code is divided in a declarations part and the rest of the program* */

```
Program = { Declarations } Starter
```

/* *variable declarations part* */

```
Declarations = "var" ident { "," ident }":" Type "init" number ";"
```

/* *the "core" of the Chrishy Language* */

```
Starter = "procedure" ident [ Arguments ] ";" { Body } "end"
          {"procedure" ident [ Arguments ] ";" { Body } "end" }

Arguments = "(" "var" ident ":" Type { "," "var" ident ":" Type } ")"
```

/* *the Body expression represents a section in a while- or "if-then-else" - block* */

```
Body = Operation
     | Condition
     | Loop
     | SemExp
     | VerrExp
```

```
/* there are three types in Chrishy: integers, semaphores and locks */

    Type = "integer"
         | "semaphore"
         | "verrou"

/* there is only one possibility to express arithmetic operations */

    Operation = ident "=" Operand Operator Operand

    Operand = ident
            | number

    Operator = "+"
             | "-"
             | "/"
             | "*"
             | "%"

/* concurrency expressions */

    SemExp = "P(" ident ")" ";"
           | "V(" ident ")" ";"

    VerrExp = "verrouiller" "." ident ";"
            | "deverrouiller" "." ident ";"

/* if - then - else */

    Condition = "if" "(" CondExp ")" "then" CausalExp [ else CausalExp ]

    CondExp = [ "!" ] ident Ops Operand [ { LogOp [ "!" ]
              ident Ops Operand } ]

    CausalExp = "{" { Body } "}"

/* the while loop */

    Loop = "while" CondExp "{" { Body } "}"
```

/* *comparing and logical operators* */

```
Ops = "=="
    | "!="
    | ">"
    | "<"
    | ">="
    | "<="

LogOp = "|"
      | "&"
```

/* *end of syntactical grammar* */

As we can see in the Syntactical Grammar, every expression is terminated with a semicolon ";". For a better understanding of how the Chrishy language works and how it is written, an example, specificially the reader / writer problem with equal priorities which is a nice demonstration above all, containing the most significant keywords follows:

```
// variable declarations
var nb_lecteurs: integer init 0;
var r, lr: semaphore init 1;
var mutex: semaphore init 1;

procedure debut_lecture;
   P(lr);
   P(mutex);
   nb_lecteurs = nb_lecteurs + 1;
   if (nb_lecteurs == 1) then {
      P(r);
   }
   V(mutex);
   V(lr);
end;

procedure fin_lecture;
   P(mutex);
   nb_lecteurs = nb_lecteurs -1;
   if (nb_lecteurs == 0) then {
      V(r);
```

```
   }
   V(mutex);
end;

procedure debut_ecriture;
   P(lr);
   P(r);
end;

procedure fin_ecriture;
   V(r);
   V(lr);
end;
```

2.3 Parser Generation

Basically, a parser converts text that can be read by humans into data structures known as parse trees, which are understood by the computer. This process is similar to compiling a Java source file (i.e. .java) into the corresponding Java bytecode (i.e. .class) that can be executed on any Java Virtual Machine. The correctness of the grammar is the key to a quick and easy generation of the parser. Unfortunately it is quite impossible to detect every illogical step made and so, such faults appear often later when it has become much harder to correct them. More to say about failure recovery in section 6 afterwards. Mentioned above, the JavaCC tool has been chosen as parser generator:

2.3.1 JavaCC - JavaCompilerCompiler

JavaCC is one of the most popular automatic parser generators for use with Java applications. JavaCC is a tool that reads a high-level grammar specification (i.e. grammar.jj) and converts it to a Java program (i.e. program.java) that recognizes matches to the grammar. In addition to the parser generator itself, JavaCC provides other standard capabilities related to parser generation such as tree building, actions, debugging, etc. Both JavaCC and the parsers generated by JavaCC may be executed on a variety of Java platforms (i.e. "Write Once, Run Anywhere").

Name and function of each of the generated Java files:

TokenMgrError.java Returns a detailed message for the error when it is thrown by the token manager to indicate a lexical error.

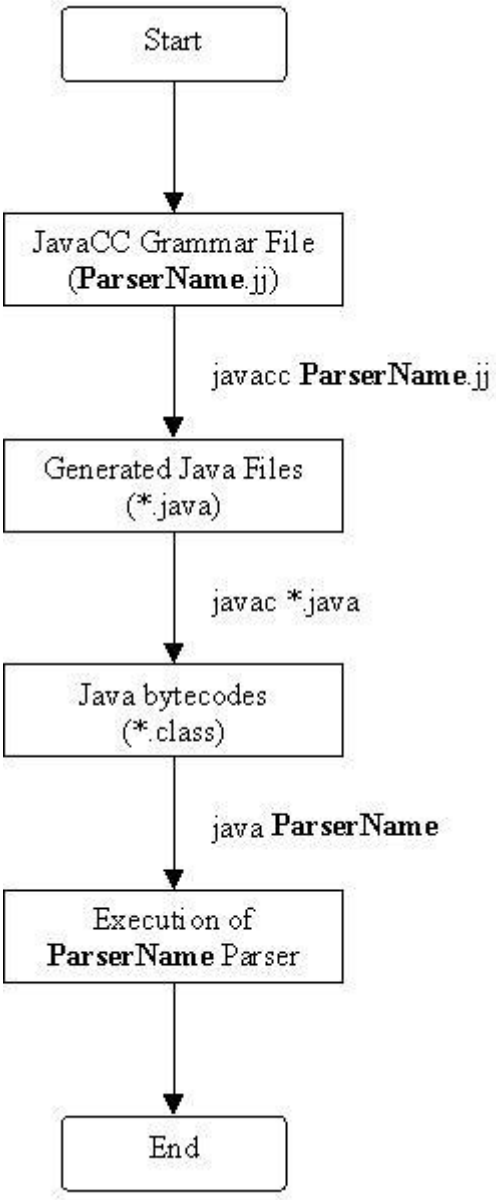

Figure 1: Flowchart illustrating the steps involved in creating a JavaCC parser program

ParseException.java This exception is thrown when parse errors are encountered.

Token.java Describes the input token stream.

ASCII_CharStream.java An implementation of interface CharStream, where the stream is assumed to contain only ASCII characters (i.e. without UNICODE processing).

ParserName.java This is the main parser program. My parser name is ChrishyParser.java

ParserNameTokenManager.java Token Manager for the parser.

ParserNameConstants.java Definition of constants for the parser.

The power of automatic parser generation is that it allows programmers to concentrate on the grammar and NOT worry about the correctness of the implementation. This can be a tremendous time-saver in both simple and complex projects. However, there are some major differences in the JavaCC grammar compared to the EBNF:

Regular Expression	Meaning
$e1 \| e2 \| e3 \| \ldots$	A choice of e1, e2, e3, etc (i.e. Logical OR).
$(X)+$	One or more occurrences of X
$(X)*$	Zero or more occurrences of X
$(X)?$	An optional occurrence of X
$[X]$	A pattern that is matched by the characters specified in X
$\sim[X]$	A pattern that matches any characters NOT specified in X

Table 1: Explanations of the JavaCC Grammar

Due to these differences, I was forced to transform the Chrishy grammar to a JavaCC grammar file. The token specifications are quite similar. It is just the notation that changes. Example:

```
TOKEN: {
     < IDENT: <LETTER> (<LETTER> | <DIGIT>)* >
     | < NUMBER: (<DIGIT>)+ >
     | < ASSIGN: "=" >
     | <SEMICOLON: ";">
     ...
       }
```

For the grammar production, we have a Java-like structure. The example for the Operations gives the idea of how it is written:

```
Operation = ident "=" Operand Operator Operand
```

is transformed to

```
void Operation() :
{}
{
<IDENT> <ASSIGN> (<NUMBER> | <IDENT>) Operator()
(<NUMBER> | <IDENT>) <SEMICOLON>
}
```

2.3.2 JTB - JavaTreeBuilder

JTB is a syntax tree builder to be used with the JavaCompilerCompiler (JavaCC) parser generator. It takes a plain JavaCC grammar file as input and automatically generates the following:

- A set of syntax tree classes based on the productions in the grammar, utilizing the Visitor design pattern.
- Two interfaces: Visitor and ObjectVisitor. Two depth-first visitors: DepthFirstVisitor and ObjectDepthFirst, whose default methods simply visit the children of the current node.
- A JavaCC grammar with the proper annotations to build the syntax tree during parsing.

New visitors, which subclass DepthFirstVisitor or ObjectDepthFirst, can then override the default methods and perform various operations on and manipulate the generated syntax tree. JTB takes the written JavaCC grammar as argument and generates the following:

- The file jtb.out.jj, the original grammar file, now with syntax tree building actions inserted.
- The subdirectory / package syntaxtree which contains a java class for each production in the grammar.
- The subdirectory / package visitor which contains Visitor.java, the default visitor interface, as well as DepthFirstVisitor.java, a default implementation which visits each node of the tree in depth-first order.

After the syntaxtree implementation in the grammar, the parser could finally be generated. However, it is mostly after this step that one comes across a variety of bugs committed in the grammar. Various tests and debugging procedures illustrated, that it is not too trivial to invent a new programming language and write its corresponding grammar. But at the end, a working parser for the Chrishy Programming Language was born, which implies: Phase I of the AVCP is accomplished.

3 Phase II: The Generation of the States

After having reached a level, where we can go and parse in the Chrishy Language, it was now time to think first about the "how" and then about the "how far" we can go in the state generation. At the beginning, a graphical state diagram was the aim. Now, after having invested a not too small amount of time and effort in the creation of the programming language itself, this aim became a little more an utopia because of the lack of time. So it was decided to go just as far as possible. In the very beginning, the concentration was focused on one single thread that executes a specific procedure of the given code. Hence, the user has to designate, which procedure this thread will execute. The implementation of more than one thread will be discussed in section 3.4.

3.1 Evaluation Concepts

There were several different concepts of how to proceed to generate the states. The first idea was to choose a limited set of breakpoints in the given code and to make the modifications only dependent of them. This method did not succeed because we need to take every little alternation of variables or expressions into consideration in order to keep the state-logics straight. Step by step the strategy evolved and at the end, the concept of an evaluation of the states via a pseudo-object environment emerged. But what does that mean? The idea was to transform the instructions given in Chrishy little by little, into so called pseudo-objects, a pseudo implementation of the corresponding instructions in Java. So, every important instruction in the code will cause an increment of the current state by one. For instance, a loop is transformed into a WhileFunction, a semaphore into a SemaphoreObject and locks become VerrouObjects. Within this object-oriented environment, the state will increase gradually. Above all, it is much easier to modify the current state by just modifying the parameters of the related object. This is the base concept that has been chosen finally.

3.2 The State Evaluation

The described concept, the pseudo-object implementation is indeed a powerful method. Unfortunately, it worked not as desired because the changes of the states require a deeper analysis of the given code. To illustrate that, we assume the following lines in the code:

```
...
if(condition) then {
   block_A
}
else{
   block_B
}
...
```

Our first algorithm would translate that into:

```
// comment              ->     no state modification
arithmetic operation    ->     no state modification
if(){}else{}            ->     state = state + 1
                               (via an IfFunction object)
```

But as we can see, the if-then-else expression would be treated as one single object. However, most of the time, then- and else-blocks will not be empty, so we have a lack of information (instructions that are ignored). So, it was decided, that EVERY instruction given - except comments - would cause an effect on the states, i.e. the alternation of variables would have to be taken into account. Hence, this modification means that the whole analysis of the states would become much more complicated because we need to "translate" the instructions given in the Chrishy Programming Language into Java, what we initially liked to omit. But nevertheless it has been done this way. The instructions from above were now handled by a new algorithm.

```
// comment              ->     no state modification
arithmetic operation    ->     state = state + 1
if(){}else{}            ->     if( condition == true ){
                                    analyze block_A }
                               else { analyze block_B }
```

So, the analysis of statements (if- and while-conditions) was another item that had to be performed. These statements are verified every time a new thread comes across of it; e.g. after every while loop. The only situation, where we decrement the state value, occurs after a second execution of such a while loop. There we have to reinstruct the same set of instructions, i.e. we have to go back to a specific state that has already been reached before and execute the same instructions again. The extent of the algorithm for the multithreaded application is described in 3.4 later on.

After this step, the project was now in a condition, where every thinkable code could be analyzed with the limitation that it is only one thread that is executing one method or one procedure.

3.3 Today's Project Progress

The one-thread solution has been reached and works perfectly and so the AVCP has evolved to a platform for a wide spectrum of applications. The multithread implementation has not been realized until now; it will be the subject of another project representing the continuation of this one. The only multithread-like feature that marks the end of this first step of the whole idea, is the implementation of a second thread with the only limitation that they run in a sequential way, i.e. the only possibility is to let them execute their parts of the code one after another. From this state forward, a real multithread application will be possible to realize, but this also will be the subject of a project that will follow.

3.4 Multithread State Analysis

State evaluations with more than one or sequentially launched threads will be a little bit more complicated. In the following, some major thoughts about the generation of a complete state diagram by presupposing a limited number of threads n that execute a specific part of the code $P(n)$. First we assume $n = 2$ and let $L(n)$ be the number of lines:

```
P(1):   instruction 1.1        P(2):   instruction 2.1
        instruction 1.2                instruction 2.2
        ...                            ...
        instruction 1.L(1)             instruction 2.L(2)
```

The algorithm first executes $P(1)$ and runs $P(2)$ sequentially. After that, step by step it will execute $P(1)$ from *instruction*1.1 to $1.(L(1) - a)$, take $2.L(1) - a + 1$ and finally $1.L(2)$ for a going from 0 to $L(1)$. Afterwards, we take $a = 2$ and perform the same procedure. This process endures until $a = L(1)$. For instance, let us consider 2 threads with 2 lines each:

```
P(1):   instruction 1.1; instruction 1.2;
P(2):   instruction 2.1; instruction 2.2;
```

This is executed in the following way:

```
1.1 / 1.2 / 2.1 / 2.2   ->   store reached states
1.1 / 2.1 / 1.2 / 2.2   ->   unify and store
1.1 / 2.1 / 2.2 / 1.2   ->   unify and store
2.1 / 1.1 / 1.2 / 2.2   ->   unify and store
2.1 / 1.1 / 2.2 / 1.2   ->   unify and store
2.1 / 2.2 / 1.1 / 1.2   ->   unify and give a pretty print of it
```

The disadvantage is that the number of possible executions does not grow linearly. See Table 2 below. And this was only a special case, when $P(1) = P(2)$ and n is fixed to 2. If we have more than two threads, we will probably need another algorithm.

# Threads	# Lines each	# Executions
1	1	1
1	n	1
2	1	2
2	2	6
2	3	20
2	4	68
3	1	6
3	2	90
...	...	...

Table 2: The number of possible executions assuming a given number of threads and an equal number of lines each

Part III
Project Structure and Tests

4 The Tool's 'Ingredients'

There are three main parts in the file structure of the tool. The first part consists of the Parser files, i.e. the files generated by JavaCC that are described in 2.3.1. The most important file for the Parser is ChrishyParser.java. The main class of my tool is a part of this first section as well. It is called GodderClass.java and stands for the leading class that handles both the user interaction and the guidance of the parsing and the state analysis. The second part is the *syntaxtree*. In this section, every grammar-related class is contained as described in 2.3.1 too. In the last part, called *visitor*, we find several different visitor patterns which represent different ways to go through the syntaxtree. A little graph describing this partitioning is found in Figure 4.

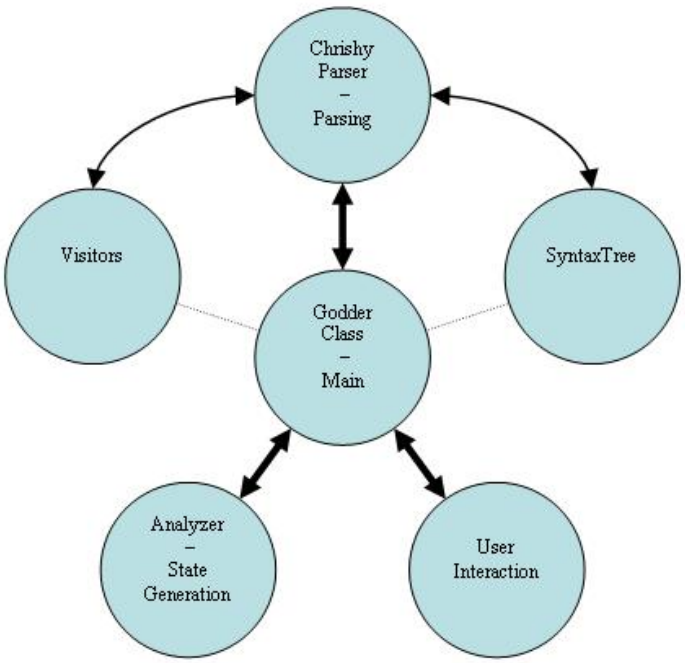

Figure 2: Graph illustrating the importance of the GodderClass and the dependencies between the most important structural elements

5 Execution Tests

Tests on our little program do not only serve us as debuggers, they also demonstrate in a clear and simple way if the program works or not. At the very end there is only the preciseness of the parsing and the state evaluation that really count. To verify the correctness of our tool, I performed several tests on the parser, i.e. the grammar. Subsequently, the state generation was taken into account which consists of the Analyzer- part and the generation of the states. The tool's output messages are shown in SmallCap fontstyle.

5.1 Tests on the Parser

There are several testfiles that contain grammatical faults. Let's verify on the programs output, if they are parsed correctly.

Testfile 1, with a faulty integer variable declaration:

```
...
var test1: integger init 2;
...
```

JTB ChrishyLanguage Printer: Reading from file test1.txt...
Encountered "integger.
Was expecting one of:
"integer" ...
"semaphore" ...
"verrou" ...

Testfile 2, containing a procedure without the demanded "end;"-expression

```
...
procedure test;
...
\\ missing end expression
```

JTB ChrishyLanguage Printer: Reading from file test2.txt...
Encountered "<EOF>.
Was expecting one of:
"end" ...
"if" ...
"while" ...
"verrouiller" ...

"DEVERROUILLER" ...
"P(" ...
"V(" ...
<IDENT> ...

Testfile 3, with an operation in the header of a while loop instead of a condition

```
...
while(a+b){...}
...
```

JTB ChrishyLanguage Printer: Reading from file test3.txt...
Encountered "+.
Was expecting one of:
">" ...
"<" ...
"<=" ...
">=" ...
"==" ...
"!=" ...

Testfile 4. This file is written correctly and should not produce errors. It contains the languages most features and finally proves that the parser works (n.b.: it doesn't make sense at all but it's just about the grammar):

```
// this file compiles. it contains most of the language's features
var semi : semaphore init 1;             // tests variable declarations
var verri : verrou init 1;
var inti, winti : integer init 3;

procedure a;
   inti = inti + 1;                      // tests diverse operations
   inti = inti * winti;

   if(inti >= winti & winti != 2) then { // tests the if-clause
      while (inti < 30 & inti != winti) {// tests multiple conditions
         P(semi);
         deverrouiller.verri;            // tests concurrency-clauses
         V(semi);
         inti = inti + 5;
      }
```

```
   }
   else { if(inti != winti) then {} else{}}
end;
```

JTB ChrishyLanguage Printer: Reading from file test4.txt...
JTB ChrishyLanguage Printer: Java program parsed successfully.

5.2 Tests on the Analyzer & the State Generator

After having proved the perfect operational sequence of the parser, we are interested in the question, if the analyzing tool also works properly. To prove this, there is a quantity of additional testfiles:

Testfile 5 demonstrating what happens, when an illegal operation occurs:

```
var  vari :  integer  init  0 ;
  procedure test ;
     vari = 1 / vari; // should generate a "division by zero"-error
  end ;
```

Proceeding with the evaluation of the states...
Current State: 0
Division by zero. Please rewrite your Inputfile. Last reached state was: 1

Testfile 6 shows that the state is incremented after each expression (here there are only semaphores). It shows also that a semaphore blocks due to it's initialisation value:

```
   var  sem :  semaphore  init  2 ;
   procedure test ;
      P( sem ) ;
      P( sem ) ;
      P( sem ) ; // the thread should be blocked here
   end ;
```

Proceeding with the evaluation of the states...
Current State: 0
Current State: 1
Current State: 2
We are blocked due to a semaphore in the following state: 3

Testfile 7 The state value is incremented by passing the operation form 0 to 1. This testfile shows also that the evaluation of the condition in the if-clause is performed accurately:

```
var counter: integer init 0;
var verr: verrou init 0;
procedure test;
   counter = counter + 1;
   if (counter == 1) then {
      verrouiller.verr; // it doesn't work if the then-block is executed
   }
   else { }
end;
```

Proceeding with the evaluation of the states...
Current State: 0
Current State: 1
We are blocked due to a lock in the following state: 2

Testfile 8, showing the state generation for a looping while structure. After the recommencing of the loop, the state value is decremented to the value of the threads first entry:

```
var counter, sillyOperator: integer init 0;
var semA: semaphore init 3;

procedure test;
   while (counter >= 0) {
      P(semA);
      sillyOperator = sillyOperator + 0; // just an operation that counts
      counter = counter + 1;             // for an additional state
   }
end;
```

Proceeding with the evaluation of the states...
Current State: 0
Current State: 1
Current State: 2
Current State: 3
Current State: 1 */* second execution of the loop */*

Current State: 2
Current State: 3
Current State: 1 /* *third execution of the loop* */
Current State: 2
Current State: 3
We are blocked due to a semaphore in the following state: 1

Testfile 9 This test is dedicated to the case when we have two threads that run sequentially through a common procedure. The state value has now an additional digit for the second thread. The evaluation will come to an end successfully.

```
var counter, sillyOperator: integer init 0;
var semA: semaphore init 5;

procedure test;
   counter = counter * 0;
   while (counter <= 1) {
      P(semA);
      sillyOperator = sillyOperator + 0;
      counter = counter + 1;
   }
end;
```

Proceeding with the evaluation of the states...
Current State: 0 0
Current State: 1 0
Current State: 2 0 /* *first thread enters loop* */
Current State: 3 0
Current State: 4 0
Current State: 2 0 /* *first thread enters loop again* */
Current State: 3 0
Current State: 4 0
Current State: 4 1
Current State: 4 2 /* *second thread enters loop* */
Current State: 4 3
Current State: 4 4
Current State: 4 2 /* *second thread enters loop again* */
Current State: 4 3
Current State: 4 4
Evaluation successful. Last significant state: 4 4

6 Encountered Difficulties and Problems

The first problems raised, when inventing the grammar for Chrishy, our own programming language. Everybody who has ever created a grammar for a language that did not exist yet, knows that this is a real source for errors: Every little missing dot, comma or semicolon can cause a very long and time-wasting search for the little bug that is often hidden behind one single pixel. Another item that is worth mentioning were the problems of compatibility between the two parser generating tools I used, i.e. JavaCC and JTB. In fact, on JavaCC there was no further development after the year 2000. So, certain methods that were generated became deprecated in meantime. This signifies I had to rewrite methods that are not meant to be rewritten. These new methods hardly interacted with JTB, so this was another big difficulty to solve. Due to the fact that I spent most of the time creating the analyzer, the large part of the complications were encountered there. However most of them were errors due to the complexity of the traduction of the Chrishy expressions into the pseudo-objects and especially into Java. The most demanding parts were the conditions. If-then-else-structures as well as while loops were the most tricky traduction hurdles. The loops were implemented by recursive calls to the analyzer, considering the number of executed intructions in the loop in order to maintain the logic of the state evaluation.

Part IV
Final Thoughts

7 Guidelines and Ideas for Following Projects

As mentioned, the multithread implementation and the state evaluation of the latter are the main targets for following projects. Additional possibilities are the development of a tool consisting of a GUI (Graphical User Interface) which decomplicates the handling with the not too trivial components of the AVCP and the extention of the Chrishy Language to one that implements even more features. However, the latter is not recommended because the whole project is based on this grammar. So changes in the rules or even new rules would cause an enormeous amount of work to do. Nevertheless the effort wold be worth it!

8 Concluding Comments

"Tempus fugit", the time flies, as the Romans used to say. What concerns me, I had the same feeling. Just began with it and the time is already over. However, the AVCP project was and is still a very interesting and exciting piece of work. And what we have reached is impressive: The parser and the analyzer are working perfectly within the implemented two-threaded environment. And the basis for future works on it is set and nothing lies on the way to a graphical representation of the states. Besides that, this project offered a better comprehension of concurrency in programming and a deeper view in Parser generators as well as the Java Programming Language.